Liechtensteinisches Gymnasium

Machine Learning and Artificial Neural Networks

Konstantin Wohlwend
7Na

Research Paper in Informatics
Academic Advisor: Daniel Oehry

15 April 2017

This is a slightly updated version (Jan 19, 2019). Appendizes A and B have been removed and replaced by a GitHub link to the source code. No other changes have been made.

Contents

1 Introduction

1.1 Machine Learning in Today's World

In the last decades, the amount of computers used in personal and professional environments has significantly increased. Machine learning is a technique that has been getting more important in the most recent years especially, because the data gathered by machines is immense and can't be mined or analyzed by humans.

Important applications of machine learning have changed our every-day life: Search engines like Google or Bing categorize our behavior online to offer personalized recommendations based on the user's and similar person's likes. Speech recognition services, like Apple's Siri or Microsoft's Cortana, learn to understand and interpret the users' voices and gradually improve every time they're talked to. In some hospital or medical clinics, symptom data of customers is analyzed and compared by computers to preemptively diagnose illnesses or diseases [1]. Computer-driven traders can assist in stock price prediction to some extent. In general, one can say that machine learning is applicable to the problems where traditional programming fails because the algorithm is too complex to code manually, while humans fail due to the sheer amount of data. Despite common belief, artificial intelligence can't magically solve problems that a traditional computer program can not. The strength of machine learning systems lies solely in the ability to automatically adapt to problems a human does not entirely comprehend.

1.2 Principles of Machine Learning

Every machine learning algorithm works in a similar fashion: It is a blackbox taking data and parameters (which could be considered the program code) as an input, then generating an output. Subsequently, the parameters will be adjusted to generate a better output, either over time or instantly. The type of the data, parameters and the output, but also architecture of the blackbox and the technique used to adjust the parameters all vary wildly on the algorithm used. Simpler algorithms may try to imitate Darwin-inspired randomized evolution, while more sophisticated ones make use of mathematical tools from statistics and calculus to find the best-fitting parameters.

Many machine learning algorithms have sought inspiration from either human or animal behavior. The ability to purposefully recognize patterns and learn from preceding scenarios was long thought to be exclusive to conscious living beings, but research in the early 80s changed that mindset [2].

1.3 Terminology: Artificial Intelligence

The term *artificial intelligence*, or *AI* for short, is heavily overloaded in the English language. One definition says artificial intelligences are conscious but artificial beings that have a human-like intelligence. In video gaming, AI is often used to refer to the code behind a *non-player character*, for example an enemy or a shopkeeper. In machine learning, artificial intelligence is a mostly not conscious computer program that acts differently based on bygone situations. Whenever this paper refers to artificial intelligence, the third definition will be used unless mentioned otherwise.

1.4 What to Expect

The ultimate goal in this paper is to create an AI that learns to play and win a simple card game with rules unknown prior to the AI starting to learn. There are three notable types of learning algorithms, two of which will be detailed in this paper:

Reinforcement Learning or RL to teach the system that winning is good whilst losing is bad. This is comparable to a human's hormones, most notably dopamine.

Supervised Learning or SL to store and process results by RL. This is comparable to the human's brain.

Unsupervised Learning or UL which has a wide variety of uses, but is useful to pre-process data before it's inputted into an SL algorithm. This is comparable to the computation done in a human's eye, which already filters signals before they even reach the nervous system. The simple card game AI created here does not need pre-processing of any kind, but as problems grow more complex unsupervised learning becomes much more important.

1.5 Conventions

In most cases, this paper uses one-based indexing when referring to vectors or matrices, but in some (specifically noted) situations zero-based indexing is more convenient. The practical parts follow Java's coding conventions and always uses zero-based indexing.

Note the difference between the absolute value/modulus $|x|$ and the euclidean norm $||x||$, where the former is the euclidean distance to the origin of a **real number** ($|a+bi| := \sqrt{a^2}$ with $a, b \in \mathbb{R}$), while the latter is the generalization over the **vector space** (euclidean length; $||v|| := \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$ with $v \in \mathbb{R}^n$). In this paper, the absolute value of a vector or matrix is an element-wise operation ($|M|_{i,j} := |M_{i,j}|$ with $M \in \mathbb{R}^{n \times m}$).

$X_{i,j}$ is the element in row i of column j of a matrix X. The ith row is denoted by $X_{i,*}$, while the jth column is denoted by $X_{*,j}$ (those are row/column vectors, respectively). Special care has to be taken for vectors – x_i could mean $x_{1,i}$ or $x_{i,1}$, depending on whether x is a column

$(n \times 1)$ or row vector $(1 \times n)$.

X^T is used to denote the transpose matrix (so that $X^T_{i,j} = X_{j,i}$).

0 may be the scalar zero, or the zero vector/matrix of any dimension. This is always clear from context, however.

The Hadamard product (also known as element-wise multiplication or Schur product) is denoted by $\bullet$ (such that $(A \bullet B)_{i,j} := A_{i,j} \cdot B_{i,j}$), while the Hadamard/element-wise power is denoted by $A^{\bullet b}$ ($\left(A^{\bullet b}\right)_{i,j} := (A_{i,j})^b$). Addition, subtraction and division are always element-wise if the least dimensional element is a scalar or the matrix dimensions match, or row-/column-wise if it is a row-/column-vector.

2 Reinforcement Learning

2.1 Idea, History and Applications

The idea behind reinforcement learning is simple. The computer, often referred to as *agent*, has a *state* (eg. the current position in a virtual world). From there, an *action* (eg. move left) will be selected and a *reward* (eg. +100 because there is a rare artifact at the new position, or −100 if there is an enemy) returned. The agent will remember the state-action-reward-triple, and if it finds itself in the same state again, it will select the action with the most promising reward.

Historically, the idea originated when analyzing animal behavior. Edward Thorndike, one of the biggest psychologists in the early 20th century, formulated a thesis in conjunction with animals long before computers existed. In the 60s, the base idea for reinforcement learning was developed by a few individuals, notably Marvin Minsky, John Andrae and Donald Michie. In the 80s, reinforcement learning experienced its biggest boom. [3]

With the huge computer structures and immense calculation power needed for decent results becoming realizable in the 21st century, reinforcement learning is now an important part of machine learning in robotics. In robot soccer, for example, scoring a goal may be a positive reward, while falling or an own goal could be a negative reward. In 2015, the Google-owned company DeepMind utilized Reinforcement Learning (along with other techniques) to beat the world-class Lee Sedol in a game of Go, who was long thought to be unbeatable by computers [4].

2.2 Q-Learning

Q-Learning is a popular reinforcement learning algorithm. The most basic version maps all rewards (*Q-values*) to a state-action pair in a so-called *Q-table*. Whenever an agent finds himself in state s_t, he will look-up the reward Q from the Q-table for each possible action a_t. Whatever Q-value is the highest will define the algorithm's next action. Now, the agent will update the Q-table based on the returned reward $r_{returned}$, the next state s_{t+1} and the next action a_{t+1}.

$$Q(s_t, a_t) := r_{returned} + Q(s_{t+1}, a_{t+1})$$

***Intention**: The Q-value is always equal to the direct reward the agent gets from doing a step, plus the reward known to exist by going further, for example a cake down the street.*

Because $Q(s_t, a_t)$ changed, and $Q(s_{t-1}, a_{t-1})$ was assigned from it, one could update all preceding Q-values as well. Care has to be taken in the implementation however, because when a Q-value for time t has been changed, the best action for $t-1$ may not remain the same. This will not change the actual result, but drastically decrease the number of iterations

needed until it is found (at least as long as the reward returned for a given state-action pair is constant).

In some cases, it makes sense to introduce a learning rate α and a discount factor γ, both in the range $(0, 1]$, defaulting to 1. The learning rate determines how quickly the algorithm learns. This is only useful if the returned reward for a given state-action-pair is not always the same. If there's no punishment (negative reward) for the agent when moving around randomly (in other words, time does not matter), then he might get stuck in a loop where he's moving in a circle for hours instead of taking the shortest way to the reward (think what an immortal, inexhaustible human would do). To fix this, one may lower the discount factor, or subtract a constant value from $r_{returned}$ every time (since that will punish the agent for moving).

$$Q(s_t, a_t) := (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_{returned} + \gamma \cdot Q(s_{t+1}, a_{t+1}))$$

***Intention - Learning rate**: If an agent previously received a cake by doing something but not this time, then he doesn't have to trash everything he learned. A learning rate of $\alpha = 0.3$ means that his new predicted reward is 30% of his newest attempt and 70% of his previous prediction.*

***Intention - Discount factor**: A discount factor of $\gamma = 0.8$ means that if the agent needs to take one extra step to get to the cake, it tastes only 80% as sweet. This is not needed if there's a punishment for wasting time.*

To make the AI less deterministic, one may add a certain chance ϵ for the prediction to be ignored entirely and randomly select an action instead. Usually, one would only keep the random chance during the training stage (especially when training the algorithm against itself), but sometimes it may make sense to leave the random factor in even after deploying the code (for example in a card game, where determinism is subject to be abused by your opponent). In the latter, $Q(s_t, a_t)$ needs to be modified to reflect the change. Note that a_t is still the action with the highest reward for state s_t, not the action that was chosen after the randomization. n_a is the number of possible actions for state s_t.

$$Q(s_t, a_t) := (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_{returned} + \gamma \cdot E(s_{t+1}, a_{t+1}))$$
$$E(s_t, a_t) = (1 - \epsilon) \cdot Q(s_t, a_t) + \epsilon \cdot \frac{1}{n_a} \cdot \sum_a Q(s_t, a)$$

***Intention - Randomization**: When training an algorithm, especially against itself, it may get stuck on a path that it considers the best even though it isn't. Randomizing the output may lead the AI to new discoveries.*

***Intention - Expected value**: Instead of the most optimistic value Q, the expected value E will be added to the reward, which is nothing more than a weighted average of all possibilities.*

***Intention - Expected value vs. Q-value**: Generally speaking, the Q-table should contain*

Q-values that will be useful once the code has been deployed. If one doesn't intend to use the random chance ϵ outside of training, the unmodified version of Q(s, a) should be used. A small ϵ-value however is useful even after finalizing the Q-table wherever human behavior should be imitated, in which case the modified reward prediction above fits better.

The initialization value for the Q-table, which can be seen as a "curiosity value", is still subject to heavy debates. Large values will cause the agent to wander around trying to find new, bigger rewards, while smaller values cause the agent to stay wherever a small reward is. In some cases, it makes sense to set it to a constant value. However, a 200-man study conducted by the Hebrew University of Jerusalem showed that, in order to get a human-like behavior, one can set the initialization values to the agent's first reward [5]. The same phenomena is abused in gambling – to keep their hopes high, slot machines bait players with high rewards when they start playing.

2.3 Exemplary Practical Implementation of Q-Learning

Appendix A contains source code for a simple Q-learning implementation written in Processing. It visualizes the algorithm by putting the agent into a virtual world with rewards and traps. After running Main.pde, a menu will appear.

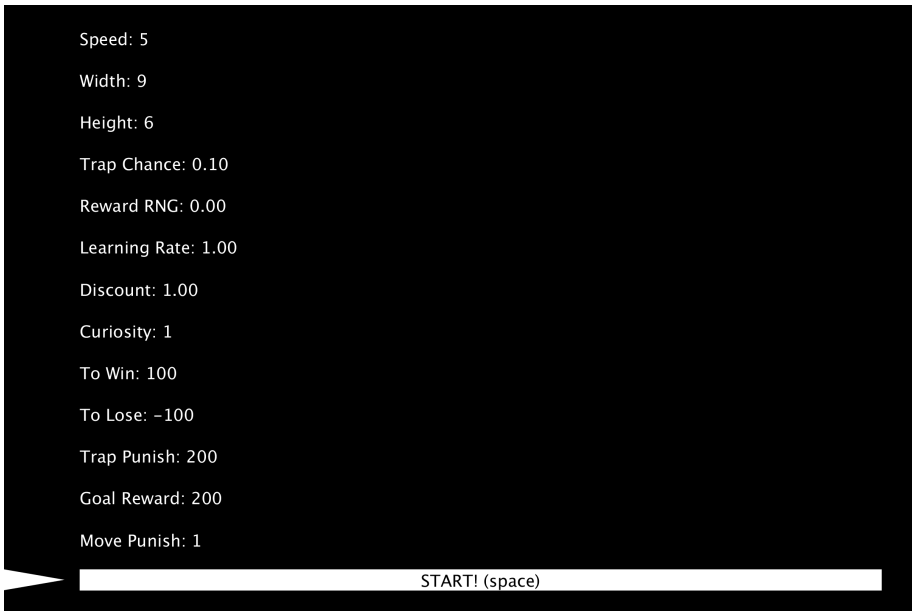

Figure 1: The main menu

One can navigate through the menu with the arrow and space keys. There are 13 customization options available:

Speed Steps per second.
Width and Height The dimensions of the virtual world, in tiles.

Trap chance Frequency of traps. If there are too many traps, it may be possible that there is no path for the agent to walk on.

Reward RNG Reward standard deviation, normally distributed.

Learning Rate, Discount and Curiosity Input variables into Q-learning as explained above.

To Win and To Lose The score needed to win or lose the game (respectively). If a game is won or lost, it will start again from the beginning.

Trap Punish and Goal Reward Amount of points deducted or added to the score when stepping on a trap or the goal (respectively).

Move Punish Amount of points deducted to the score on every step.

By moving the triangle to Start using the arrow keys and pressing space, one can start the simulation.

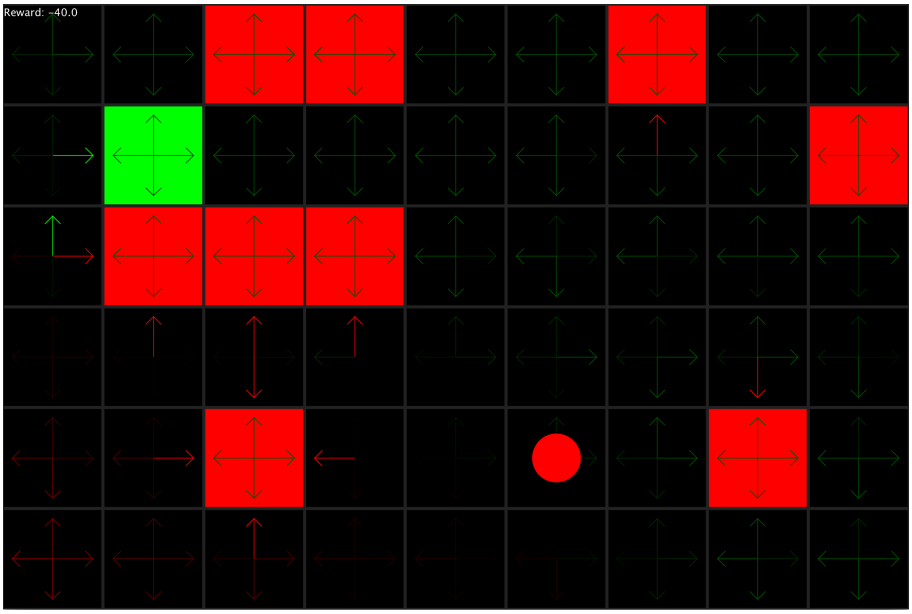

Figure 2: An agent on his way to traversing a currently mostly unexplored area on the right after already having explored the bottom left part. Even though a way to the goal has been found (denoted by the light green arrows), he does not currently know how to reach it from his current position.

Since the agent stops exploring once it finds a goal whose reward is higher than the curiosity reward, it may get stuck on a slow path. Usually, this is not an issue and also a part of human and animal behavior, but if it turns out to be problematic, one may increase ϵ and/or the curiosity value.

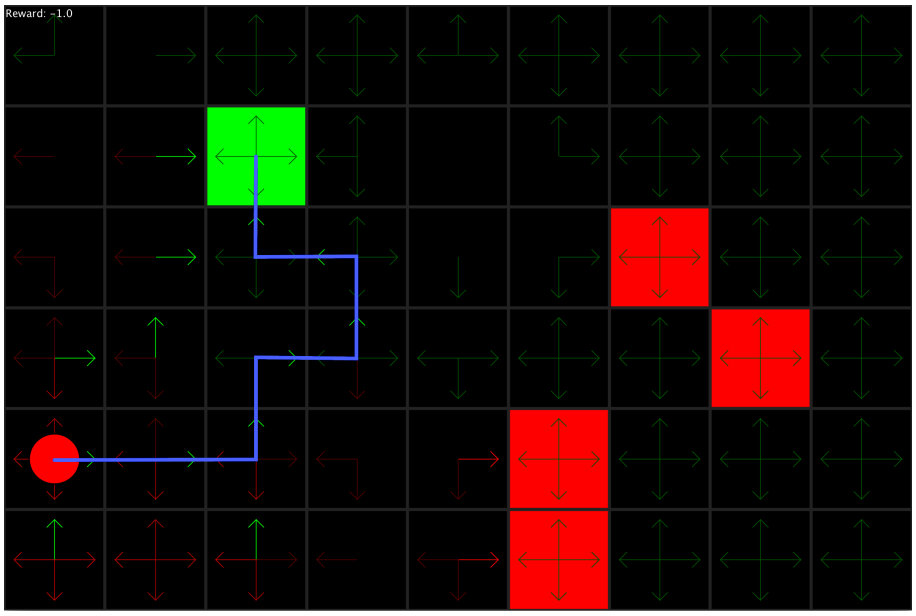

Figure 3: The agent found a path to the goal that is not the fastest (in blue). It will never try any other paths again.

3 Supervised Learning

3.1 Idea, History and Applications

Supervised learning is one of the, if not the one, most important techniques in machine learning. Similarly to an array or a look-up table, the algorithm takes an input x and returns an output y. Unlike the traditional methods however, supervised learning algorithms can predict values that were not inserted into the system before. This works because the algorithm tries to guess a function h (known as *hypothesis*) so that $y \approx h(x)$.

Supervised learning was partly inspired by regression analysis techniques, but also by the human brain. The former originated in the 19th century, when Gauss and Legendre independently needed it to analyze astronomical data [6]. Over the course of the following two-hundred years, it has become an important part of statistics. Applying this knowledge to develop self-teaching algorithms however, even if first (theoretical) steps were taken in the 50s, has not happened until the 1980s. For a very long time, those algorithms have co-existed with traditional programming as a neat extra – but recent rise of processing power in the 21st century that made even a sub-average mobile phone able to compute huge amounts of data and the popularity of online services made machine learning one of the most important subjects in computer science.

Whenever a large amount of input data x and output data y is available, but there is some more data for which there's only an input, supervised learning can be used to predict the output. This is useful in trend estimation, pattern and speech recognition, medical centers, marketing, spam/fraud detection, and of course, in Q-learning as an alternative for the Q-table, which is what the practical part needs supervised learning for.

3.2 Linear Regression

3.2.1 General Idea

Linear Regression is an important part of statistics, often used for interpolation of data. However, in computer science, it has a special meaning because many supervised learning algorithms derive from it. Linear regression tries to find the best fit for the parameters θ and b of the linear hypothesis $h(x)$ to approximate the output y using the input x:

$$h(x) = \theta \cdot x + b$$

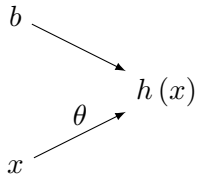

Figure 4: A visualization of a simple linear regression algorithm.

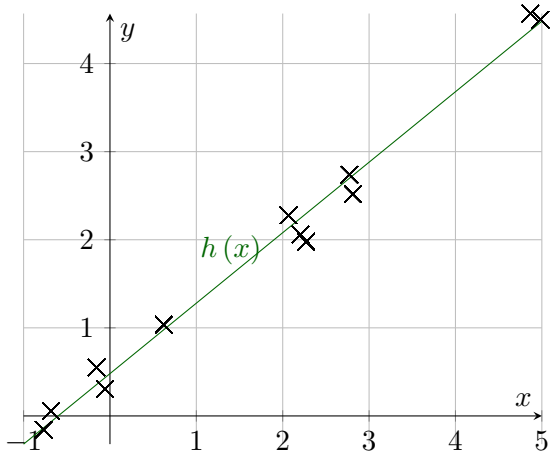

Figure 5: Several output values y plotted against the input x and an example hypothesis $h(x)$ with $\theta = 0.8$ and $b = 0.5$.

Since the hypothesis is a linear function, linear regression in this form only works on linear data sets (see Figure 5). However, later in this paper, more advanced algorithms will be detailed that are not restricted by this limitation.

If there are n input variables (so-called *features*), a row vector (which is a $1 \times n$ matrix, unlike the more common column vectors) may be used for x, and a column vector for θ:

$$\begin{aligned} h(x) &= \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \ldots + \theta_n \cdot x_n + b \\ &= x \cdot \theta + b \end{aligned}$$

***Intention**: Linear regression deals with multiple inputs by summing the individual outputs. This is equal to the matrix multiplication of the row vector x and the column vector θ, or a scalar multiplication of the two vectors.*

To unite all parameters in one vector θ, define $\theta_0 := b$ and $x_0 := 1$. This indexing interferes with the one-based indexing convention made, but it simplifies things in the long run because it simplifies the vectorized variant: $h(x) = x \cdot \theta$ (since $x_0 \cdot \theta_0 = b$).

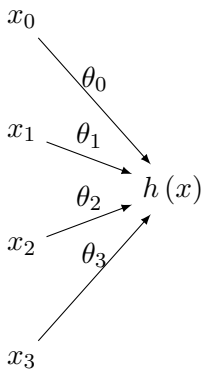

Figure 6: A linear regression system with 3 features x_1 to x_3 and a bias term x_0.

$$\begin{array}{cccc} X & \cdot & \Theta & = & h(X) \\ \begin{bmatrix} 1 & 7 & 4 \end{bmatrix} & \cdot & \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} & = & 15 \end{array}$$

Figure 7: Exemplary linear regression with 3 inputs (including the bias term).

With not only multiple input features, but also multiple output nodes that are stored in a row vector $y \in \mathbb{R}^{1 \times m}$, θ becomes a matrix of the dimensions $n \times m$ (where n is the number of features including the bias term, and m the number of output nodes). To comply with common conventions, upper-case letters will be used to denote matrices (Θ in this case). Applying matrix multiplication on a $1 \times n$ input row vector x with an $n \times m$ parameter matrix Θ returns the correct $1 \times m$ row vector $h(X) = X \cdot \Theta$.

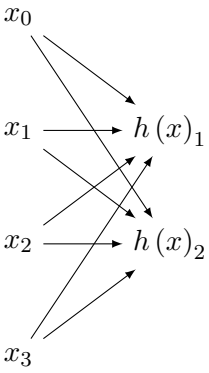

Figure 8: A linear regression system with 3 features (and bias) and 2 outputs.

Because an input and output data set usually consists out of multiple examples however, the in- and output row vectors may be expanded to $c \times n/c \times m$-matrices, where c is the set size (number of examples).

$$
\begin{array}{ccccc}
X & \cdot & \Theta & = & h(X) \\
\begin{bmatrix} 1 & 7 & 4 \\ 3 & 0 & 0 \\ 3 & 4 & 4 \\ 4 & 8 & 2 \\ -1 & -2 & 3 \end{bmatrix} & \cdot & \begin{bmatrix} 1 & 2 \\ 2 & 2 \\ 0 & 1 \end{bmatrix} & = & \begin{bmatrix} 15 & 20 \\ 3 & 6 \\ 11 & 18 \\ 20 & 26 \\ -5 & -3 \end{bmatrix}
\end{array}
$$

Figure 9: Exemplary in-, parameter and output matrices. The linear regression system used takes 3 inputs (two features and one bias) and returns 2 outputs. There are 5 examples (rows) in the data set.

Putting it all together, linear regression tries to find the best-fitting parameter matrix Θ for the following approximation:

$$
\begin{aligned}
&Y \approx h(X) \\
&h(X) = X \cdot \Theta \\
&X \in \mathbb{R}^{c \times n};\ h(X) \in \mathbb{R}^{c \times m};\ \Theta \in \mathbb{R}^{n \times m}
\end{aligned}
$$

... where X is the input matrix, $h(X)$ the predicted output, c the data set size, n the input count including the bias term, and m the output count.

In theory, there is nothing limiting supervised learning to matrices with elements in $\mathbb{R}$. These could be complex numbers or even elements of a non-numeric set (eg. colors). In practice however, due to a computer's nature of working well with real numbers, as long as no special hardware is built it makes little sense to generalize the algorithm.

Because the most optimistic output Y and the input X are known, one could think the parameters Θ can be found by looking at the problem as a linear system. However, this is not the case, because the elements of X and Y may have a statistical deviation from the "perfect" results, meaning that no exact parameters Θ exist for all X and Y. This is where cost functions become important.

3.2.2 Cost Functions

In order to improve the parameter matrix Θ, a function J is needed to measure the error. In statistics, J is called the *loss function*, however, in machine learning it is better known as *cost function*. The return value is a row vector where the jth element is the error for the jth

output (jth column of Y). The most intuitive cost function is a linear one:

$$\begin{aligned} J\left(X,Y,\Theta\right)_j &= \frac{1}{c}\sum_{k=1}^{c}\left(\left|Y-h\left(X\right)\right|_{k,j}\right) \\ &= \frac{1}{c}\sum_{k=1}^{c}\left(\left|Y-X\cdot\Theta\right|_{k,j}\right) \\ X\in\mathbb{R}^{c\times n};\ Y,\ h\left(X\right)&\in\mathbb{R}^{c\times m};\ \Theta\in\mathbb{R}^{n\times m};\ J\in\mathbb{R}^{1\times m} \end{aligned}$$

(Note: Usage of the element-wise absolute value notation as explained in the introduction)

***Intention**: $X\cdot\Theta$ is the predicted output by the algorithm for the input X and the parameters Θ. Its difference to the actual output can be used to measure the parameter error. This is now done for all examples in the data set, and averaged.*

A cost function like this, however, has multiple flaws:

- Not continuously differentiable – even though a quasi-derivative could be defined, this causes problems with some algorithms used later on.
- Even though the FABS processor instruction for calculating the absolute value has been heavily optimized on most hardware [7], most software implementations are very inefficient (eg. Java's).
- If one training example has a difference of 5 between predicted and actual output, the cost function punishes it as 5 different training examples had a difference of 1, even though the latter is usually a far better estimation.

The simplest way to fix all of these issues is to square the difference. This will get rid of the abstract value, and therefore the function will be continuously differentiable, while also punishing higher deviations from the optimal value more severely. It is common practice (but in no way necessary) to add $\frac{1}{2}$ as a factor to the function. This way, the exponent will cancel out in the derivatives. The new function shares some visual similarities with the formula for the standard deviation.

$$\begin{aligned} J\left(X,Y,\Theta\right)_j &= \frac{1}{2c}\sum_{k=1}^{c}\left(\left(Y-X\cdot\Theta\right)_{k,j}\right)^2 \\ &= \frac{1}{2c}||\left(Y-X\cdot\Theta\right)_{*,j}||^2 \end{aligned}$$

***Intention**: Another way to explain the formula is to represent the actual output $Y_{*,j}$ and the predicted output $\left(X\cdot\Theta\right)_{*,j}$ as points in a multi-dimensional euclidean space. The distance between these is a good measure for the error. Since the numeric output is irrelevant and the cost function is only used to compare ("which parameters are better"), one can freely square (to get rid of the root) and multiply the term with $\frac{1}{2c}$.*

3.2.3 Error Minimization

Now that the problem is clear and a cost function has been formulated, there needs to be a way to minimize it. All somewhat logical solutions to this problem need the partial derivative of the cost J with respect to the parameters Θ:

$$\begin{aligned}\frac{\partial J_j}{\partial \Theta_{i,j}} &= \frac{1}{2c}\frac{\partial}{\partial \Theta_{i,j}}\left(\sum_{k=1}^{c}\left((Y - X \cdot \Theta)_{k,j}\right)^2\right) \\ &= \frac{1}{2c}\sum_{k=1}^{c} 2\left(Y - X \cdot \Theta\right)_{k,j} \cdot \frac{\partial}{\partial \Theta_{i,j}}(Y - X \cdot \Theta)_{k,j} \\ &= \frac{1}{c}\sum_{k=1}^{c} X_{k,i}\left(X \cdot \Theta - Y\right)_{k,j}\end{aligned}$$

This derivative can be represented as a matrix $\frac{\partial J}{\partial \Theta}$ of the same size as Θ where the element (i, j) is the derivative of J_j with respect to $\Theta_{i,j}$.

$$\frac{\partial J}{\partial \Theta} = \frac{1}{c} \cdot X^T\left(X \cdot \Theta - Y\right)$$

To find the absolute minimum, this derivative can be equated with 0 and solved for Θ, as that will return the extremum of the cost function (which has to be a minimum).

$$\begin{aligned}\frac{1}{c} \cdot X^T\left(X \cdot \Theta - Y\right) &= 0 \\ X^T\left(X \cdot \Theta - Y\right) &= 0 \\ X^T X \cdot \Theta - X^T Y &= 0 \\ X^T X \cdot \Theta &= X^T Y \\ \left(X^T X\right)^{-1} X^T X \cdot \Theta &= \left(X^T X\right)^{-1} X^T Y \\ \Theta &= \left(X^T X\right)^{-1} X^T Y\end{aligned}$$

However, because they work better with more complex algorithms, and because even the optimized Coppersmith-Winograd algorithms' complexity for inverting matrices is fairly high with $O\left(n^{2.373}\right)$ [8], iterative optimization algorithms are more common in practice. The simplest one to understand and implement is *gradient descent*. It works by starting with randomized parameters Θ (each element should be small but random, for example generated by a gaussian distribution with the variance 0.01), then taking a small step towards the local minimum with every iteration:

$$\Theta := \Theta - \alpha \cdot \frac{\partial J}{\partial \Theta}$$

***Intention**: For a small learning rate α, $\alpha \cdot \frac{\partial J}{\partial \Theta}$ is positive if J grows with Θ. To decrease J, $\alpha \cdot \frac{\partial J}{\partial \Theta}$ must be subtracted from Θ.*

α is the learning rate and one of the most important constants in supervised learning. If the

value is too big, Θ overshoots and the local minimum will never be found, but if it's too small, the algorithm learns too slowly. Plotting J after a few iterations against multiple test values of α can help finding the perfect learning rate. In contrast to the learning rate in Q-learning, in supervised learning α may be greater than 1.

There are two variants of gradient descent. The first is stochastic gradient descent, which uses the algorithm described above. Batch gradient descent, on the other side, splits X and Y into small batches row-wise (so that the amount of columns stays the same) and applies the algorithm on each of these independently. The latter usually finds a local minimum in less iterations because Θ gets updated not only after the entire calculation, but already during the process. Theoretically, a batch size of 1 would be optimal, but in practice, parallelized hardware (eg. GPUs, multi-core processors, computer networks) can deliver a better performance if the batches are bigger.

One huge downside of most iterative algorithms is that most of them are unable to find the absolute (or global) minimum. In linear regression, this does not matter because there may only be a single minimum – however, keep it in mind as you use more complex learning algorithms.

Other iterative algorithms like the *conjugate gradient method* (better known as CG) or the *Broyden-Fletcher-Goldfarb-Shanno algorithm* (BFGS) can find the local minima faster than gradient descent, but are much harder to understand and implement. They also can't be used with online learning systems where the algorithm adapts to a continuous stream of data, which will be elaborated and used later.

3.2.4 Feature Engineering

Feature Scaling

When using gradient descent to minimize the error J, one has to pick a learning rate α. This value depends on multiple factors, an important one being the standard deviation σ of the features (inputs) and outputs. However, if the deviation from the mean of the individual in- and outputs are too different from each other, then one would need multiple learning rates. Instead, one can simply normalize the average standard deviation before the data gets fed into the regression system (here only with X, but if there are multiple outputs the same should be done with Y as well):

$$
\begin{aligned}
X_{scaled} &= \frac{X - \bar{X}}{\sigma} \\
\bar{X} &= \frac{1}{m} \sum_{k=1}^{m} (X_{k,*}) \\
\sigma &= \sqrt{\frac{1}{m} \sum_{k=1}^{m} \left(X_{k,*} - \bar{X} \right)^{\bullet 2}}
\end{aligned}
$$

If the values $\bar{X}$ and σ change, so will the parameters Θ that fit $h(X) \approx Y$ the best. Therefore, these two values should be calculated once with some training set, and not modified by any new data that enters the system. In other words, the very first input data used to train the regression system should also define $\bar{X}$ and σ for all other input sets inputted later on.

If the actual outputs Y of the training sets are scaled, the system also returns scaled output predictions $h_{scaled}(X)$. Getting the unscaled values back is simple:

$$\begin{aligned} Y_{scaled} &= \frac{Y - \bar{Y}}{\sigma} \\ \implies h(X) &= h_{scaled}(X) \bullet \sigma + \bar{Y} \end{aligned}$$

Non-Linear Regression

Linear regression can only model linear functions. If the data is not on a linear function, then adding non-linear features may help. That means, say there is a feature x which is known to influence the output in a non-linear way, adding a new feature x^2 may help. If this is done for polynomials up to some order k, then even a normal linear regression algorithm will learn to model the Taylor series of the function. If there are two features x_1 and x_2, one might even add new, combined features like $x_1 x_2$ or $x_1^2 x_2$. These features are usually created externally before they're fed into the regression system.

However, high-order polynomials can cause the input matrix to become very large and clunky. If there are 5 non-polynomial features and one wants to add polynomial and combined features up to x^5, there will be $5^6 = 7776$ features. Polynomial regression should be utilized with caution. Even though it *can* adapt the parameters to almost all data sets, it is not an efficient solution. Plotting J against k (or the number of total features) may help with the decision of whether, and if so, how many polynomials to use, not only in linear regression, but also in more complex supervised learning algorithms detailed later.

Overfitting

If there are many non-linear features but not enough data, the optimization algorithm might find minima for J that are "too good" in a sense where the linear regression fits a complicated function through every single example in the training set while examples outside of the training set are predicted incorrectly.

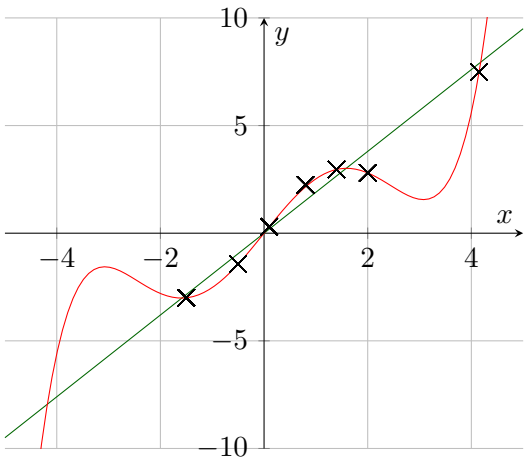

Figure 10: Due to a lack of data (eg. between $x = 2$ and $x = 4$), non-linear regression managed to find the red complicated hypothesis with the use of polynomial features. While this cost function fits those specific points almost perfectly, in most use cases the green hypothesis would be a much better fit to the data.

In order to both prevent and diagnose overfitting (and potentially even underfitting), the data set is usually split in three parts:

Training set Often two or three times as big as each of the other, the training set is the main set, and should be used when training parameters only (for example using gradient descent).

Validation set The validation set is used to plot J against constants like the learning rate or order of polynomials. If you use the training set to do these tasks, you will not plot the algorithm's capability to predict new examples, but rather the algorithm's capability to adjust its parameter matrix Θ.

Test set The test set is a third part of the data set. It is solely used to evaluate the regression learner. It is completely independent from any parameter optimization or constant choices and is therefore a good, unbiased estimate to measure how well the trained algorithm will perform when entirely new examples will be added to the algorithm. If the data set is very small, the test set is often merged with the validation set, but if possible those two should be separate from each other. Only use the test set to find out how well the algorithm performs. **Do not tune anything based off the test set error!**

(Note: If the data set is sorted, remember to shuffle it before splitting it into pieces.)

If there's a low cost when calculated with the training set but not with the validation set, it is a clear sign of overfitting. One way to reduce overfitting is to keep parameters small (and therefore preventing non-linear regression from using too many non-linear features unless really needed). One can simply add a *weight regularization term* to J:

$$J\left(X, Y, \Theta\right)_j = \frac{1}{2c} \sum_{k=1}^{c} \left(\left(Y - X \cdot \Theta \right)_{k,j} \right)^2 + \lambda \cdot \frac{1}{2} \sum_{k=1}^{n} \left(\Theta_{k,j} \right)^2$$

λ is the *weight regularization parameter* and controls how much the algorithm should be punished if its parameters are big. To find a good value, plot J (using the validation set) against λ. Note how Θ's first row has the index 0, but the weight regularization term starts summing at $i = 1$. This is because the bias term does not contribute to overfitting and therefore doesn't need to be regularized either. This means a conditional is needed in the derivative for the very first row:

$$\frac{\partial J_j}{\partial \Theta_{i,j}} = \frac{1}{c} \cdot X^T \left(X \cdot \Theta - Y \right) + \begin{cases} 0, & \text{if } i = 0 \\ \lambda \cdot \Theta_{i,j}, & \text{otherwise} \end{cases}$$

Another, more effective way to deal with overfitting is to get more training data and therefore preventing the algorithm from finding an alternative hypothesis in the first place.

3.3 Classification Problems

3.3.1 Logistic Regression

Not all problems can be solved with a real-valued output. Classification problems are choices – for example separating spam from serious mails, diagnosing potential illnesses for a patient with certain symptoms, classifying iris plants based on their dimensions or identifying objects on a picture. For that reason, there is a technique called logistic regression. It is mostly the same as linear regression, but there's one fundamental difference: It only outputs values in a certain range.

The activation function $f(z)$ is a function that maps all output values to values in a specific range (known as *activation*), often $(0, 1)$ or $(-1, 1)$. A common one is the S-shaped logistic or sigmoid function which goes towards 0 for $z \to -\infty$, and towards 1 for $z \to +\infty$ (which could mean no and yes, respectively):

$$\begin{aligned} f(z) &= \frac{1}{1 + e^{-z}} \\ f'(z) &= \frac{\mathrm{d}}{\mathrm{d}z} \frac{1}{1 + e^{-z}} \\ &= -\left(1 + e^{-z}\right)^{-2} \cdot \left(-e^{-z}\right) \\ &= \frac{e^{-z}}{\left(1 + e^{-z}\right)^2} \\ &= \frac{1}{e^z \cdot \left(1 + \frac{2}{e^z} + \frac{1}{e^{2z}}\right)} \\ &= \frac{1}{e^z \cdot \left(\frac{1 + 2e^z + e^{2z}}{e^{2z}}\right)} \\ &= \frac{e^z}{\left(1 + e^z\right)^2} \end{aligned}$$

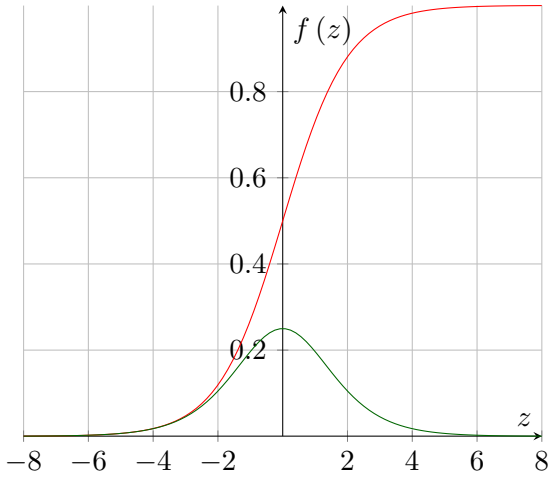

Figure 11: The sigmoid activation function $f(z)$ (in red) and its derivative $f'(z)$ (in green).

A quick check shows that $f'(z)$ is equal to $f(z) \cdot (1 - f(z))$, which can be useful in practice for optimization purposes.

The hypothesis of logistic regression is the same as the one used in linear regression, but wrapped in the activation function:

$$h(X) = f(X \cdot \Theta)$$

This way, the hypothesis will only output values between 0 and 1. Wherever linear regression would output a high value, logistic regression with the sigmoid activation function outputs a value close to 1, and wherever linear regression would output a negative value, logistic regression with the sigmoid activation function outputs a value close to 0. If the former are interpreted as a "yes" and the latter as a "no", the regression system can now learn from simple classification data ("is the e-mail spam or not?").

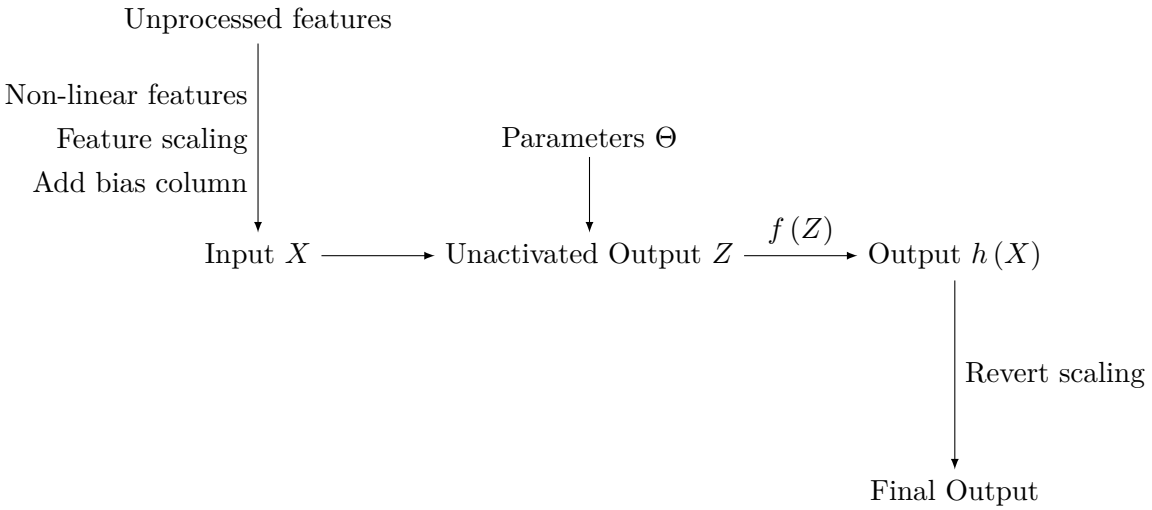

Figure 12: A visualization of the steps taken in the logistic regression algorithm.

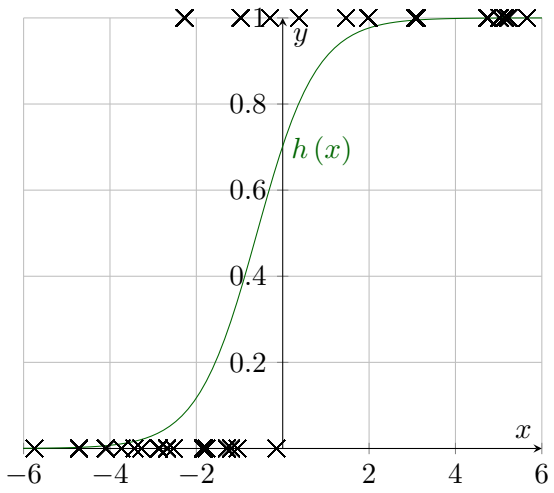

Figure 13: Logistic regression data visualized. The output on the y-axis, which is 1 for yes and 0 for no, is plotted against the input on the x-axis. The bias term is $\theta_0 = 0.86$, parameter for the first (and only) feature is $\theta_1 = 1.44$. The algorithm predicts an almost certain yes for $x > 2$, and an almost certain no for $x < -4$.

Since the hypothesis has changed, the cost function needs to be updated. In theory, one could simply replace the old hypothesis with the new one, but since logistic regression is about classification, it makes more sense to create a new cost function with the following properties:

- Continuously derivable
- Cost of an entirely wrong prediction (ie. predicting $0.0000\ldots$ when the answer is yes) is infinite
- Cost of a perfectly right prediction (ie. predicting $1.0000\ldots$ when the answer is yes) is 0
- If two predictions are not entirely wrong or entirely right (ie. one predicting 0.5, the other predicting 0.7 when the answer is yes), then the closer one has a smaller cost (0.7 in the example)

A logarithmic cost function fulfills all of these:

$$J\left(X, Y, \Theta\right)_j = -\frac{1}{c}\sum_{k=1}^{c}\left(Y_{k,j} \cdot \ln\Big(h\left(X\right)_{k,j}\Big) + \left(1 - Y_{k,j}\right) \cdot \ln\Big(1 - h\left(X\right)_{k,j}\Big)\right)$$

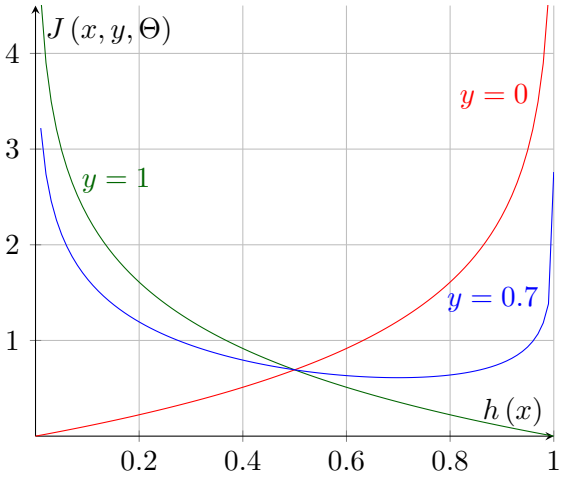

Figure 14: The logarithmic cost function used for classification problems. Green curve for actual output $y = 1$, red curve for $y = 0$, blue curve for average actual output $y = 0.7$ (in case the output depends on a random or unknown factor).

Calculating the derivative is a bit more involved:

$$\begin{aligned}
\frac{\partial J_j}{\partial \Theta_{i,j}} &= \frac{\partial}{\partial \Theta_{i,j}} \left(-\frac{1}{c} \sum_{k=1}^{c} \left(Y_{k,j} \cdot \ln\left(h\left(X\right)_{k,j} \right) + \left(1 - Y_{k,j}\right) \cdot \ln\left(1 - h\left(X\right)_{k,j} \right) \right) \right) \\
&= -\frac{1}{c} \sum_{k=1}^{c} \left(\frac{\partial}{\partial \Theta_{i,j}} \left(Y_{k,j} \cdot \ln\left(h\left(X\right)_{k,j} \right) + \ln\left(1 - h\left(X\right)_{k,j} \right) - Y_{k,j} \cdot \ln\left(1 - h\left(X\right)_{k,j} \right) \right) \right) \\
&= -\frac{1}{c} \sum_{k=1}^{c} \left(\left(\frac{Y_{k,j}}{h\left(X\right)_{k,j}} - \frac{1}{1 - h\left(X\right)_{k,j}} + \frac{Y_{k,j}}{1 - h\left(X\right)_{k,j}} \right) \cdot \frac{\partial}{\partial \Theta_{i,j}} h\left(X\right)_{k,j} \right) \\
&= -\frac{1}{c} \sum_{k=1}^{c} \left(\frac{Y_{k,j} \cdot \left(1 - h\left(X\right)_{k,j} \right) - h\left(X\right)_{k,j} + Y_{k,j} \cdot h\left(X\right)_{k,j}}{h\left(X\right)_{k,j} \left(1 - h\left(X\right)_{k,j} \right)} \cdot \frac{\partial}{\partial \Theta_{i,j}} f\left(X \cdot \Theta\right)_{k,j} \right) \\
&= -\frac{1}{c} \sum_{k=1}^{c} \left(\frac{Y_{k,j} - Y_{k,j} \cdot h\left(X\right)_{k,j} - h\left(X\right)_{k,j} + Y_{k,j} \cdot h\left(X\right)_{k,j}}{h\left(X\right)_{k,j} \left(1 - h\left(X\right)_{k,j} \right)} \cdot f'\left(X \cdot \Theta\right)_{k,j} \cdot \frac{\partial}{\partial \Theta_{i,j}} \left(X \cdot \Theta\right)_{k,j} \right) \\
&= -\frac{1}{c} \sum_{k=1}^{c} \left(\frac{Y_{k,j} - h\left(X\right)_{k,j}}{f\left(X \cdot \Theta\right)_{k,j} \left(1 - \left(f\left(X \cdot \Theta\right)_{k,j} \right) \right)} \cdot f\left(X \cdot \Theta\right)_{k,j} \left(1 - f\left(X \cdot \Theta\right)_{k,j} \right) \cdot X_{k,i} \right) \\
&= -\frac{1}{c} \sum_{k=1}^{c} \left(Y_{k,j} - h\left(X\right)_{k,j} \right) \cdot X_{k,i} \\
&= \frac{1}{c} \sum_{k=1}^{c} X_{k,i} \left(h\left(X\right) - Y \right)_{k,j}
\end{aligned}$$

Surprise or no surprise, the derivative eventually turns out to be the exact same that was used in linear regression (just with a different hypothesis). Now that the hard part is done, getting the vectorized version of the derivative is less of a hassle:

$$\frac{\partial J}{\partial \Theta} = \frac{1}{c} \cdot X^T \left(h\left(X\right) - Y \right)$$

Gradient descent still works the same. The algebraic approach, however, looks slightly different. First, $f(z)$ solved for z is needed.

$$\begin{aligned} f(z) &= \frac{1}{1+e^{-z}} \\ f(z) \cdot (1+e^{-z}) &= 1 \\ e^{-z} &= \frac{1-f(z)}{f(z)} \\ z &= -\ln\left(\frac{f(z)}{1-f(z)}\right) \\ f^{-1}(z) &= -\ln\left(\frac{z}{1-z}\right) \end{aligned}$$

Now, the derivative must be equated with 0.

$$\begin{aligned} \frac{1}{c} \cdot X^T (f(X \cdot \Theta) - Y) &= 0 \\ X^T (f(X \cdot \Theta) - Y) &= 0 \\ f(X \cdot \Theta) - Y &= 0 \\ f(X \cdot \Theta) &= Y \\ X \cdot \Theta &= f^{-1}(Y) \\ \left(X^T X\right)^{-1} X^T X \cdot \Theta &= \left(X^T X\right)^{-1} X^T \cdot f^{-1}(Y) \\ \Theta &= \left(X^T X\right)^{-1} X^T \cdot f^{-1}(Y) \end{aligned}$$

3.3.2 Multiclass Classification

Not all problems are simple yes-or-no questions. Sometimes, there are multiple classes of which only one is correct, for example when classifying an animal species, or identifying objects on a picture. This problem can be dealt with in two ways.

The first, One-vs-One, splits the classes into small groups of two, where each group gets its own output node. An output value of 0 now represents one class, 1 the other (as if one were to ask "Is it rather A than B?" for the classes A and B, and the reply is either yes or no). This has the big upside of keeping the number of outputs low (and potentially even inputs, even if not elaborated here these two techniques can also be used to feed classification data into a learner system) but if the classification is not always obvious, it may cause problems – say there are four classes A, B, C and D, where A, B and C, D are the two groups – in One-vs-One, "it is most likely A or B" has the same output as "it is neither A nor B" (both 0.5). One-vs-All, on the other side, assigns four individual outputs, one for each class. It is equivalent to asking "Is it A?" for each class.

3.4 Neural Networks

3.4.1 History

Artificial neural networks could be considered the "kings" of supervised learning. Even though computationally expensive and hard to understand, they have proven to work well in many scenarios and can even imitate human behavior.

In 1958, Frank Rosenblatt created a supervised learning algorithm called *Perceptrons* which is considered to be what started artificial neural network research. In practice, a single perceptron is nothing else than logistic regression with a rather simple activation function [10]:

$$f(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$$

When connecting multiple perceptrons, they can perform simple tasks like addition or subtraction. However, artificial neural networks were long thought to be unable to learn more complex operations using traditional algorithms (eg. gradient descent), especially the logical XOR operation [11].

This rapidly changed when Paul Werbos re-invented the backpropagation algorithm that originated in the 60s and applied it to neural networks in 1974 and 1982 [12]. The fading interest suddenly came back, partly because hardware improved as well. As the millennium approached, new algorithms, especially for recurrent neural networks, were found. *Long short-term memory*, presented in 1997 by Jürgen Schmidhuber and Sepp Hochreiter, is one of the most popular supervised learning techniques up until today [13]. With the rise of online services in the last decade, huge neural networks have become as important as never before. Customers of online shops are not served by humans, but rather by computers needing to learn how to fit the client's needs. In 2016, a neural network managed to beat Lee Sedol in a match of Go, who was long thought to be unbeatable for computers [14].

3.4.2 General Idea

Neural networks are in fact only connected layers of logistic regression, where each layer's output is the next one's input (note that a column with only ones is still added for the bias term). Simple problems like $\neg$ (NOT), $\wedge$ (AND) or $\vee$ (OR) gates don't need more than just normal logistic regression, but in order to represent $\oplus$ (XOR) at least two layers are needed (since $A \oplus B = C \wedge D$ with $C = \neg(A \wedge B)$ and $D = A \vee B$, one layer has to process C and D first).

The parameters Θ are called *layer weights* $\Theta^{(l)}$ of a layer l in neural networks. The dimensionality of these matrices may be different for each layer, in other words, each layer may have

a different amount of in- and output nodes (with the only restriction that one layer's input node count must be equal to the last layer's output node count plus 1 for the bias term). $Z^{(l)}$ is used to denote the unactivated output of a layer i, while $A^{(l)}$ is the output after passing it through the activation function. This output is added to a column with only ones, which results in the next layer's input $I^{(l+1)}$. So, for n layers, the hypothesis may look like this:

$$
\begin{aligned}
I^{(1)} &= X \\
Z^{(l)} &= I^{(l)} \cdot \Theta^{(i)} \\
A^{(l)} &= f\left(Z^{(l)}\right) = f\left(I^{(l)} \cdot \Theta^{(l)}\right) \\
I^{(l+1)} &= \begin{bmatrix} 1 & \\ 1 & \\ \vdots & A^{(l)} \\ 1 & \end{bmatrix} \\
h\left(X\right) &= A^{(n)}
\end{aligned}
$$

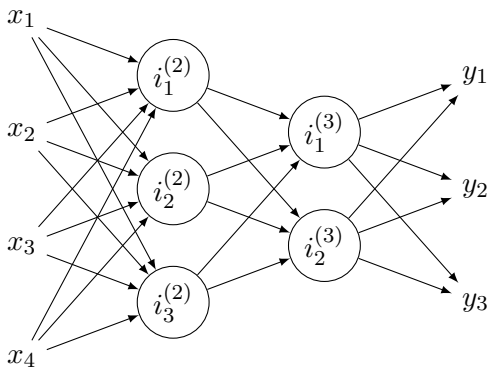

Figure 15: A neural network with four features, three outputs and two so-called *hidden layers* in-between. The input, hidden and output nodes are also known as *neurons*. It does not have bias for simplicity. (This specific network is not useful in practice because the second hidden layer is smaller than both the preceding and the output layer, which will often result in data loss)

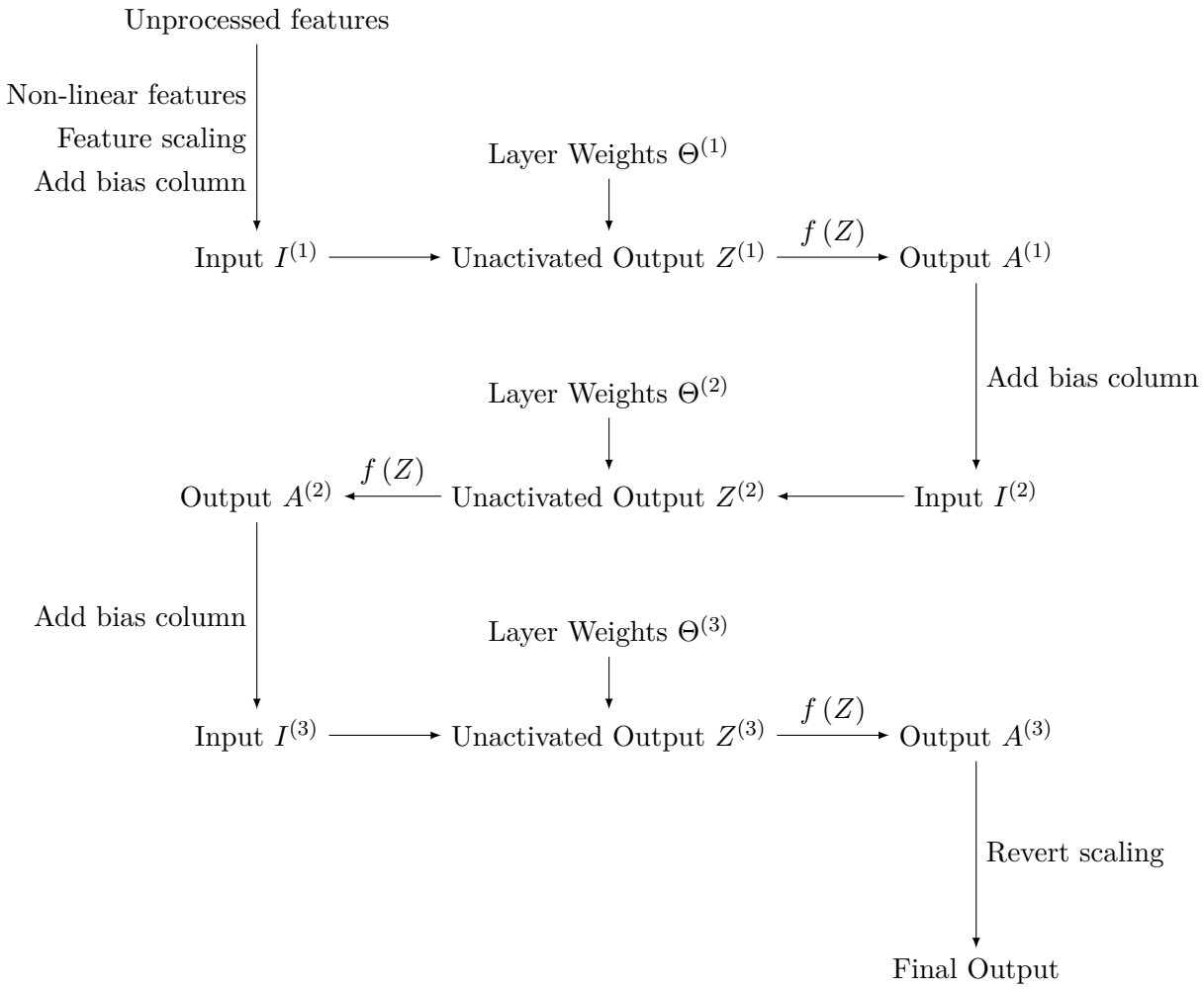

Figure 16: A visualization of the steps taken in a neural network with three layers.

The choice of a good activation function for neural networks is a lot harder. In fact, any derivable non-linear function is a potential candidate. The sigmoid function, as used in logistic regression, has proven to work well in classification but also in regression problems (note that the output must be scaled into the range $(0, 1)$). However, if the neural network consists out of many layers, some iterative learning algorithms (eg. gradient descent) can be very slow due to a problem known as *vanishing gradients*. Since the sigmoid activation function "squashes" its input values into the range $(0, 1)$, a layer output's standard deviation from the mean will always be smaller than the input's. If there are many, many layers, the standard deviation will slowly approach 0, resulting in the last layers unable to learn since the differences between the inputs are so small.

To fix this for large networks, an activation function needs to be used that doesn't squash values (or at least not on both sides). A report published in *Nature* (which refers to another paper [16]) suggests the rectified linear unit function:

At present, the most popular non-linear function is the rectified linear unit (ReLU),

> which is simply the half-wave rectifier $f(z) = \max(z, 0)$. In past decades, neural nets used smoother non-linearities, such as $\tanh(z)$ or $1/(1 + \exp(-z))$, but the ReLU typically learns much faster in networks with many layers, allowing training of a deep supervised network without unsupervised pre-training. [15]

While this function still squashes negative values (even more than the sigmoid function, since $f(x)$ is effectively squashed to 0 for $x \leq 0$), positive values will not be affected by vanishing gradient. However, if a single neuron's weights somehow reach a value where the input of the ReLU function is always negative (usually through weight initialization, but sometimes during training), it will die and not be able to learn anymore. While weight decay (especially if applied on every weight, including the usually-excluded bias column) will often automatically battle this problem, the algorithm should regularly check for dead neurons, and if found, slowly adjust the layer weights over multiple iterations (eg. by increasing the bias weight). During the later stages of training, neurons may also die if they're not needed for an almost perfect approximation (because the dataset is too simple). In that case, it is safe to remove them.

Even though it is not continuously derivable, one can define $f'(0) := 0$ and get:

$$f(z) = \max(z, 0)$$

$$f'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$$

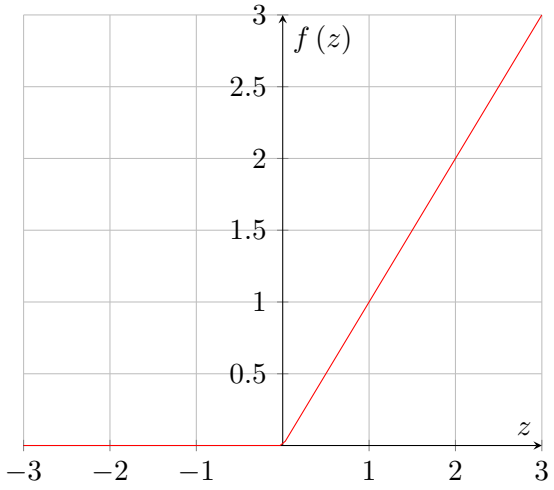

Figure 17: The ReLU activation function $f(z)$.

The rectified linear unit activation function may return any non-negative value. It can therefore not be used in classification problems. It is, however, possible to use the sigmoid activation function only for the very last layer or layers – this will maintain the fast learning of the ReLU activation function in deep networks, while also outputting values in the range $(0, 1)$.

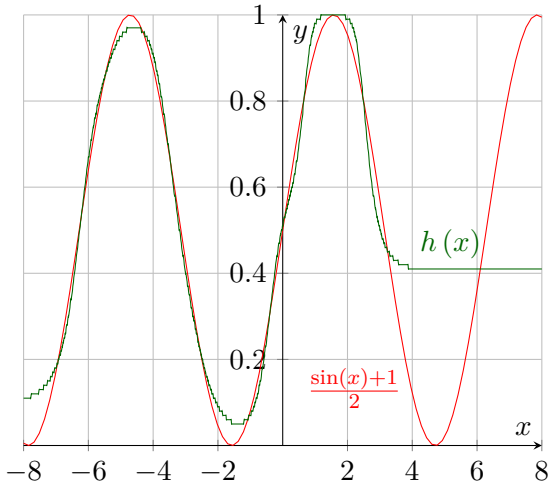

Figure 18: A neural network hypothesis (in green) trying to imitate the sine function (in red). It has two hidden layers, with three neurons each, and uses the sigmoid activation function. If the hidden layers were bigger, the function could be approximated almost perfectly.

3.4.3 Error Minimization

Generally speaking, the cost function to be used depends on the data. Notably, one can't use the logarithmic cost function with the rectifier activation function since the former expects values in the range $(0, 1)$. In case of the sigmoid activation function, however, both the logarithmic and half-square cost functions work well. Both (and most other) cost functions average each column, which makes the introduction of a "partial cost function" D very convenient. (Note that the examples below do not include weight decay, which should be applied even to neural networks)

$$J\left(X, Y, \Theta\right)_j = \frac{1}{c} \sum_{k=1}^{c} D\left(X, Y, \Theta\right)_{k,j}$$

$$D_{sq}\left(X, Y, \Theta\right) = \frac{1}{2} \left(Y - h\left(X\right)\right)^{\bullet 2}$$

$$\frac{\partial D_{sq}}{\partial h} = h\left(X\right) - Y$$

$$
\begin{aligned}
D_{log}\left(X, Y, \Theta\right) &= -Y \bullet \ln(h\left(X\right)) + (1-Y) \bullet \ln(1-h\left(X\right)) \\
\frac{\partial D_{log}}{\partial h} &= -\left(\frac{Y}{h\left(X\right)} + \frac{Y-1}{1-h\left(X\right)}\right) \\
&= \frac{h\left(X\right) - Y}{h\left(X\right) - h\left(X\right)^{\bullet 2}}
\end{aligned}
$$

Figure 16 visualizes how a neural network is a series of chained functions. In order to derivate the cost function with respect to the parameters for gradient descent, one can apply the chain rule, for example here for the output layer:

$$
\begin{aligned}
\frac{\partial J_j}{\partial \Theta_{i,j}^{(n)}} &= \frac{1}{c}\sum_{k=1}^{c}\left(\frac{\partial D_{k,j}}{\partial h_{k,j}}\frac{\mathrm{d}h_{k,j}}{\mathrm{d}A_{k,j}^{(n)}}\frac{\mathrm{d}A_{k,j}^{(n)}}{\mathrm{d}Z_{k,j}^{(n)}}\frac{\partial Z_{k,j}^{(n)}}{\partial \Theta_{i,j}^{(n)}}\right) \\
&= \frac{1}{c}\sum_{k=1}^{c}\left(\frac{\partial D}{\partial h_{k,j}} f'\left(Z_{k,j}^{(n)}\right) I_{k,i}^{(n)}\right)
\end{aligned}
$$

For simplicity, $\frac{\partial D_{k,j}}{\partial Z_{k,i}^{(l)}}$ (for the output layer, $j = i$; for any other layer, see the note below) will be denoted as *erroneousness* $\zeta_{k,i}^{(l)}$. Given the layer l, it can be determined by stepping back in Figure 16 with the chain rule:

$$
\begin{aligned}
\zeta_{k,j}^{(n)} &= \frac{\partial D_{k,j}}{\partial Z_{k,j}^{(n)}} \\
&= \frac{\partial D_{k,j}}{\partial h_{k,j}}\frac{\mathrm{d}h_{k,j}}{\mathrm{d}A_{k,j}^{(n)}}\frac{\mathrm{d}A_{k,j}^{(n)}}{\mathrm{d}Z_{k,j}^{(n)}} \\
&= \frac{\partial D}{\partial h_{k,j}} \cdot f'\left(Z_{k,j}^{(n)}\right) \\
\zeta_{k,i}^{(l-1)} &= \sum_{j=1}^{j_{max}} \zeta_{k,j}^{(l)}\frac{\partial Z_{k,j}^{(l)}}{\partial Z_{k,i}^{(l-1)}} \\
&= \sum_{j=1}^{j_{max}} \zeta_{k,j}^{(l)}\frac{\partial Z_{k,j}^{(l)}}{\partial I_{k,i}^{(l)}}\frac{\mathrm{d}I_{k,i}^{(l)}}{\mathrm{d}A_{k,i}^{(l-1)}}\frac{\mathrm{d}A_{k,i}^{(l-1)}}{\mathrm{d}Z_{k,i}^{(l-1)}} \\
&= \sum_{j=1}^{j_{max}} \zeta_{k,j}^{(l)}\Theta_{i,j} \cdot f'\left(Z_{k,i}^{(l-1)}\right)
\end{aligned}
$$

(Note: Because $Z^{(l)} = I^{(l)} \cdot \Theta^{(l)}$, $I^{(l)}$ (and therefore $Z^{(l)}$) affects all outputs of the layer, weighted by the layer weights. By visualizing the algorithm with Figure 15, it quickly becomes obvious that to calculate ζ, one has to sum the derivatives of $D_{k,*}$.)

With this erroneousness ζ, $\frac{\partial J}{\partial \Theta^l}$ needed for gradient descent is nothing more than a multiplication:

$$
\frac{\partial J_j}{\partial \Theta_{i,j}^{(l)}} = \zeta_{i,j}^{(l)} \cdot \frac{\partial Z_{i,j}^{(l)}}{\partial \Theta_{i,j}^{(l)}}
$$

In combination with gradient descent, this so-called *backpropagation* algorithm can be used to minimize errors in any layer of a neural network. The algorithm can (and should) be vectorized.

$$\begin{aligned}
\Theta^{(l)} &:= \Theta^{(l)} - \alpha \cdot \frac{\partial J}{\partial \Theta^{l}} \\
\frac{\partial J}{\partial \Theta^{l}} &= \left(I^{(l)}\right)^{T} \cdot \zeta^{(l)} \\
\zeta^{n} &= \frac{\partial D}{\partial h} \bullet f'\left(Z^{(n)}\right) \\
\zeta^{(l-1)} &= \left(\zeta^{(l)} \cdot \left(\Theta^{(l)}\right)^{T}\right) \bullet f'\left(Z^{(l-1)}\right)
\end{aligned}$$

(Note: When computing $\zeta^{(l-1)}$, remember to ignore the first column (which is the bias) of $\zeta^{(l)}$.)

***Intention**: The erroneousness ζ represents a layer's output error. The last layer's error is simply the value returned by the cost function's derivative, corrected with the activation function's derivative. A layer l's error is its "guilt" on the next layer's erroneousness, in other words, $\zeta(l+1)$ multiplied with the weight, then again corrected with the activation function's derivative.*

A problem with gradient descent arises when the ReLU function is used: If a neuron's weights reach a value (for whatever reason) where it will output 0 for any input in the training set/previous layer (example with one input: the input is in the range $[0, 10]$ and the weight is -1), the neuron will die (= become unable to change) because the activation function's derivative will always be 0. While these kinds of neurons may sometimes automatically get resurrected by a change in the input range, they often have to be fixed manually, eg. by a procedure that increases the bias weights (over multiple iterations) of any neuron whose output is almost always 0. The sigmoid activation function, or any other function whose derivative never reaches (only approaches) zero does not experience this problem.

3.4.4 Incremental Neural Networks

Online learning is a part of supervised learning where there's a constant stream of training data, eg. in a recommender system where customers constantly click on products, or in a card game where the algorithm steadily battles itself. *Incremental neural networks* work no different than classic neural networks. The only difference is the way they are trained - instead of trying to find the local minimum with a given training set, it runs a given number (if there's enough data, then one) of iterations of an optimization algorithm (eg. gradient descent) with a specific training example, then discards it and repeats with the next.

Incremental neural networks are extremely useful when used in conjunction with Q-learning, assuming there is enough data to constantly feed the algorithm. If there are many different states, it becomes very hard to store all of them in an array or a database. Incremental neural

networks can learn and predict Q-values without having to directly store them.

3.5 Supervised Learning in Practice

3.5.1 Initializing the Extermination

Rachel is a sneaky artificial intelligence. What might seem to be a little girl is in the process of conquering all of the multiverse to finally reign over humanity. But first, Rachel has to learn how to play a simple card game against the boredom she'll encounter after having extinguished all non-artificial forms of life.

The card game played in this part is very simple. Rachel receives a bunch of poker cards, of which she'll choose and discard one every turn. Next, Rachel will draw a new card *(not necessarily from the same stack of 52 poker cards, so it's theoretically possible to hold the same card multiple times)*, and win if all the cards' colors match, or get another turn if they don't. It is now her job to win the game in as few turns as possible. She knows absolutely nothing about these rules, and has never touched poker cards before.

Due to the unknown-rules nature of this problem, Q-learning seems perfect. However, if Rachel plays with more cards in her hand, the Q-table will quickly exceed the amount of resources Rachel wants to spend on a card game. Instead, a neural network will be used to store Q-values. Deep neural networks can easily support thousands of cards, but for simplicity's sake only two each 50-neuron-sized hidden layers will be used.

As a programming language, Java was chosen due to its flexibility to run on many different computer architectures and operating systems. For efficient matrix computations, the Nd4j library was used due to its wide range of features and adaptivity to large data sets. While not strictly required (but highly suggested), Maven was chosen as a build system to automagically manage dependencies. The main class of the card game is `com.n2d4.rachel.main.RachelThrow`, but `com.n2d4.rachel.main.Rachel` can also be used as a main class, though learning other things instead.

Behind the scenes, a few utility classes in the package `com.n2d4.rachel.util` will be used. These will not be included or elaborated in this paper, but are available in the downloadable program code. They are similar to libraries like Apache Commons or Google Guava and are not related to machine learning in any way.

3.5.2 Using the Machine Learning Classes

There are several classes used for machine learning purposes in Rachel's code, most notably `NeuralNetwork` and `QLearner`. They fulfill what their names promise and do the respective algorithm's jobs.

`NeuralNetwork` is a sub-class of `SupervisedLearner` (which also serves as a super-class to `LinearRegression` and `LogisticRegression`) and takes a `DataSet` (while usually a `StaticDataSet`, it may be an `OnlineDataSet` for online/incremental learning) along with the discussed parameters and each one optional `CostFunction` (default sigmoid), `ActivationFunction` (default logarithmic) and `OptimizationFunction` (default gradient descent) for its constructor. Once done, it can be trained with its `#train(...)` method, evaluated with `#getTrainingError(...)`, `#getValidationError(...)`, `#getError(...)` and used with `#process(...)`.

`QLearner` can only be used in conjunction with a `QTable`. Since Rachel combines neural networks with Q-learning, a `NeuralQTable` is perfect for the occasion. The class `NeuralQLearner` (extends `QLearner`) will take care of the neural network and the neural Q-table. The code shares a lot of similarities with the `QLearner` class in the Q-learning exemplary Processing implementation, and therefore also shares its usage: `#getBestAction(...)` will find the most promising action for a given state, `#registerReward(...)` will register a reward after a specific action has been chosen, and `#resetRound(...)` will reset the agent's current state. If `#resetRound(...)` is not called when the agent's state has been changed from the outside, the new state will be remembered as a consequence to the last action from the previous state.

A `NeuralQTable` feeds the state into the `NeuralNetwork` as its input (in Rachel's case, a One vs. All classification of the card colors) and takes the Q-values for each action from its output (one output per possible action). If the game is played with 3 cards in the hand, the neural network will have 12 inputs (four for every card) and 3 outputs (deciding which card will be discarded).

The big majority of data classes extend `VectorizedData`, a matrix wrapper class. If one wants to extract data from any of the sub-classes, they can use the static method `VectorizedData.getINDArray(VectorizedData data)`, which will return the backing Nd4j array. Any changes made to it will be reflected on the wrapper, which is why caution should be taken when doing so on data that isn't supposed to be modified.

3.5.3 Three Cards in Hand

The game is the easiest to play if the agent has three cards in hand (with only two, there are no strategies to learn). If all three card colors are different from each other which card is thrown away does not matter. If two of the three colors match, the AI should learn to discard the third. Whenever Rachel does something right, she'll instantly get feedback (throwing the third card away results in a win 25% of the time). After relatively few games played, Rachel already plays the game very well, usually winning in around 10 turns:

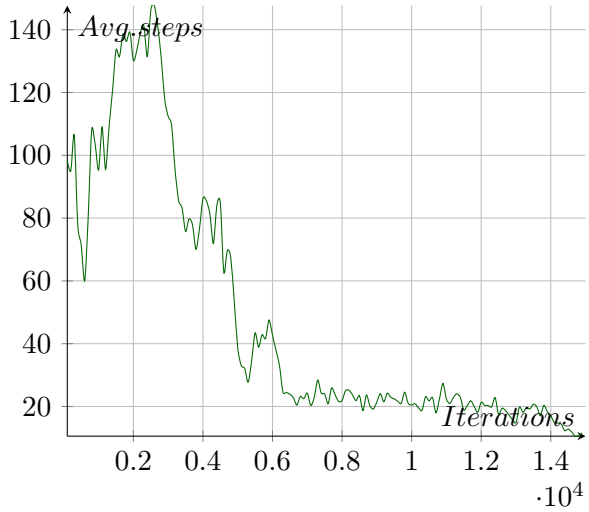

Figure 19: Average step count (weighted average where the newest result weighs 0.01; second newest result weighs $0.01 \cdot 0.99 = 0.099$ etc.) plotted against number of games played. After 1 million games played, only 7 steps are needed on average. Note that the learning rate was intentionally chosen to be low for visualizable data.

For the trained algorithm, a game may look like this:

```
Cards: [CLUBS NINE, SPADES QUEEN, HEARTS THREE]

Turn 1
Discard: 2 (HEARTS THREE)
New cards: [CLUBS NINE, SPADES QUEEN, DIAMONDS TEN]
Reward: 0.0

Turn 2
Discard: 2 (DIAMONDS TEN)
New cards: [CLUBS NINE, SPADES QUEEN, DIAMONDS TWO]
Reward: 0.0

Turn 3
Discard: 2 (DIAMONDS TWO)
New cards: [CLUBS NINE, SPADES QUEEN, HEARTS JACK]
Reward: 0.0

Turn 4
Discard: 2 (HEARTS JACK)
New cards: [CLUBS NINE, SPADES QUEEN, DIAMONDS KING]
Reward: 0.0

Turn 5
Discard: 2 (DIAMONDS KING)
New cards: [CLUBS NINE, SPADES QUEEN, HEARTS FIVE]
Reward: 0.0

Turn 6
Discard: 2 (HEARTS FIVE)
```

```
New cards: [CLUBS NINE, SPADES QUEEN, DIAMONDS JACK]
Reward: 0.0

Turn 7
Discard: 2 (DIAMONDS JACK)
New cards: [CLUBS NINE, SPADES QUEEN, SPADES TWO]
Reward: 0.0

Turn 8
Discard: 0 (CLUBS NINE)
New cards: [HEARTS ACE, SPADES QUEEN, SPADES TWO]
Reward: 0.0

Turn 9
Discard: 0 (HEARTS ACE)
New cards: [HEARTS FOUR, SPADES QUEEN, SPADES TWO]
Reward: 0.0

Turn 10
Discard: 0 (HEARTS FOUR)
New cards: [CLUBS FOUR, SPADES QUEEN, SPADES TWO]
Reward: 0.0

Turn 11
Discard: 0 (CLUBS FOUR)
New cards: [DIAMONDS TWO, SPADES QUEEN, SPADES TWO]
Reward: 0.0

Turn 12
Discard: 0 (DIAMONDS TWO)
New cards: [SPADES SEVEN, SPADES QUEEN, SPADES TWO]
Reward: 1.0
```

For the first six turns, the algorithm gets unlucky and does not get any pairs. On turn 7, it discards the third card and gets a second Spades card. The trained AI recognizes that and starts discarding the only card that doesn't match the pattern in slot 1 (the Clubs Nine) until all of the cards are Spades on turn 12.

3.5.4 Four Cards in Hand

Playing with four cards in hand is a lot harder. With three cards, the agent always got an immediate reward – however, now, it is clever not to discard cards of a color you already have twice, even though you won't win directly if you don't. Fortunately, the Q-learning algorithm takes care of that by discounting future rewards (if the discount factor γ is greater than 0). However, learning from the discount value is a lot slower than learning from an immediate reward.

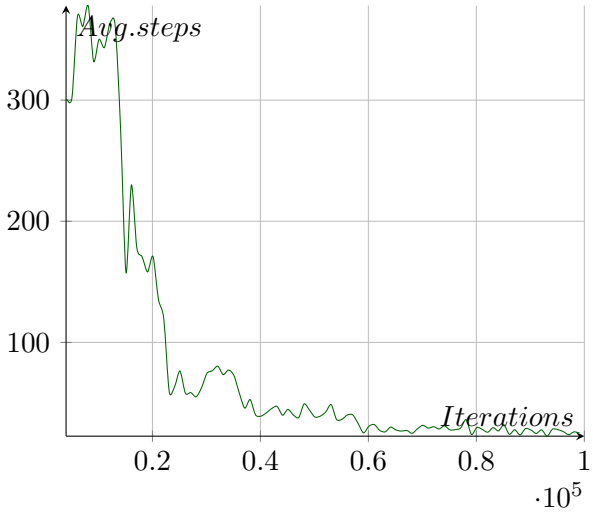

Figure 20: The drop in average steps after the network learns what to do with triples puts the average steps needed to approx. 50-75, but it takes tens of thousand additional iterations until the AI learns what to do before.

Just like an agent in a virtual world, Rachel will be satisfied with the very first half-decent way to win. In the example above, she noticed a very simple way to win was to just discard everything that's not a Clubs card:

```
Cards: [DIAMONDS TEN, HEARTS NINE, HEARTS FIVE, CLUBS ACE]

Turn 1
Discard: 0 (DIAMONDS TEN)
New cards: [DIAMONDS ACE, HEARTS NINE, HEARTS FIVE, CLUBS ACE]
Reward: 0.0

Turn 2
Discard: 0 (DIAMONDS ACE)
New cards: [DIAMONDS TEN, HEARTS NINE, HEARTS FIVE, CLUBS ACE]
Reward: 0.0

Turn 3
Discard: 0 (DIAMONDS TEN)
New cards: [SPADES FIVE, HEARTS NINE, HEARTS FIVE, CLUBS ACE]
Reward: 0.0

Turn 4
Discard: 0 (SPADES FIVE)
New cards: [DIAMONDS NINE, HEARTS NINE, HEARTS FIVE, CLUBS ACE]
Reward: 0.0

Turn 5
Discard: 0 (DIAMONDS NINE)
New cards: [SPADES THREE, HEARTS NINE, HEARTS FIVE, CLUBS ACE]
Reward: 0.0

Turn 6
```

```
Discard: 0 (SPADES THREE)
New cards: [CLUBS FIVE, HEARTS NINE, HEARTS FIVE, CLUBS ACE]
Reward: 0.0

Turn 7
Discard: 1 (HEARTS NINE)
New cards: [CLUBS FIVE, SPADES FOUR, HEARTS FIVE, CLUBS ACE]
Reward: 0.0

Turn 8
Discard: 1 (SPADES FOUR)
New cards: [CLUBS FIVE, DIAMONDS SIX, HEARTS FIVE, CLUBS ACE]
Reward: 0.0

[...]

Turn 74
Discard: 3 (DIAMONDS EIGHT)
New cards: [CLUBS JACK, CLUBS NINE, CLUBS FIVE, DIAMONDS THREE]
Reward: 0.0

Turn 75
Discard: 3 (DIAMONDS THREE)
New cards: [CLUBS JACK, CLUBS NINE, CLUBS FIVE, DIAMONDS SIX]
Reward: 0.0

Turn 76
Discard: 3 (DIAMONDS SIX)
New cards: [CLUBS JACK, CLUBS NINE, CLUBS FIVE, SPADES ACE]
Reward: 0.0

Turn 77
Discard: 3 (SPADES ACE)
New cards: [CLUBS JACK, CLUBS NINE, CLUBS FIVE, CLUBS SEVEN]
Reward: 1.0
```

Even though the AI was thrown out of context by the randomizing variable ϵ (which was set to 0.05) a few times in this specific example (explaining why the fourth slot isn't Clubs anymore at the end, and also why Clubs Five in slot 1 was replaced by Clubs Jack), it is a solid and simple strategy that the AI learned to play with no prior knowledge about the game or whatsoever.

4 Conclusion

Machine learning is booming. Adaptive computers can solve problems where humans struggle, serve customers cheaper than any employee and understand data better than anyone else. Reinforcement, supervised and unsupervised learning build the fundament of services that we have been provided with by manufacturers of our smartphones and creators of our favorite websites. In this paper, two powerful algorithms, Q-learning and neural networks, have been analyzed and implemented into a card game AI.

Nevertheless, the ever-growing topic of machine learning has still not been remotely fully covered. Unsupervised learning alone covers hundreds of algorithms that have become an important part in everyone's daily routines. Support vector machines, a supervised learning algorithm that has fallen out of fashion with the rise of neural networks, remain untouched. Recurrent neural networks like Long Short-Term Memory (LSTM) have overtaken the leading role in neural network popularity due to their ability to adapt to far more complex problems and learn from timed data.

Just as LSTM, the human brain is a recurrent neural network as well, but far more complex and larger. However, according to Intel, the exascale milestone could already be reached in 2018 [17] [18], which is also the dimension of a human's brain processing power. Stepping into the world of machine learning is stepping into a world where work is done by machines while picking the fruits is left to the humans who built them. It is stepping into a world where laziness is not bad but encouraged, and where trivial jobs have been replaced by creativity and innovative spirit. It is stepping into a world where science can flourish further because it fixed its own problems. That is, of course, if we survive our own deeds. And at least for the time being it does not seem like we will.

5 Appendix A/B: Source Code

This chapter initially contained source code for the QLearner and Rachel. However, since they're now available on GitHub, I have removed it from here. You can still access the original source code on the GitHub repository below.

`https://www.github.com/N2D4/ml-ann`

6 Bibliography

[1] Hodson, H. *Revealed: Google AI has access to huge haul of NHS patient data.* New Scientist, 29 April 2016. <https://www.newscientist.com/article/2086454-revealed-google-ai-has-access-to-huge-haul-of-nhs-patient-data>

[2] Langley, P. *The changing science of machine learning.* Published in: *Machine Learning Volume 82.* New York, Berlin et. al: Springer-Verlag, March 2011, 275-279.

[3] Sutton, R., Barto A. *Reinforcement Learning: An introduction.* Massachusetts: The MIT Press, 1998, 16-21.

[4] Wikipedia, The Free Encyclopedia *Lee Sedol.* Wikipedia, 30 January 2017. <https://en.wikipedia.org/w/index.php?title=Lee_Sedol&oldid=762718092>

[5] Shteingart, H. et al. *The Role of First Impression in Operant Learning.* Published in: *Journal of Experimental Psychology: General Volume 142.* Washington: American Psychological Association, 2013, 476-488.

[6] Wikipedia, The Free Encyclopedia *Regression Analysis History.* Wikipedia, 20 January 2017. <https://en.wikipedia.org/w/index.php?title=Regression_analysis&oldid=760969489#History>

[7] Agner, F. *Instruction tables. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.* 1 December 2016. <http://www.agner.org/optimize/instruction_tables.pdf>

[8] Wikipedia, The Free Encyclopedia *Computational complexity of mathematical operations. Matrix algebra.* Wikipedia, 5 November 2016. <https://en.wikipedia.org/w/index.php?title=Computational_complexity_of_mathematical_operations&oldid=747912668#Matrix_algebra>

[9] Rosenblatt, F. *The Perceptron: A probabilistic model information storage and organization in the human brain.* Published in: *Psychological Review Volume 65.* 1958, 386-408

[10] Wikipedia, The Free Encyclopedia *Perceptron. Definition.* Wikipedia, 30 December 2016. <https://en.wikipedia.org/w/index.php?title=Perceptron&oldid=757434768#Definition>

[11] Wikipedia, The Free Encyclopedia *Artificial Neural Network. History.* Wikipedia, 12 January 2017. <https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=759687416#History>

[12] Wikipedia, The Free Encyclopedia *Backpropagation. History.* Wikipedia, 11 January 2017. <https://en.wikipedia.org/w/index.php?title=Backpropagation&oldid=759489307#History>

[13] Wikipedia, The Free Encyclopedia *Long short-term memory.* Wikipedia, 17 December 2016. <https://en.wikipedia.org/w/index.php?title=Long_short-term_memory&oldid=755252953>

[14] Google DeepMind *AlphaGo* <https://deepmind.com/research/alphago/>

[15] LeCun, Y. et al. *Deep learning*. Published in: *Nature Volume 521*. MacMillan Publishers Ltd, 28 May 2015, 438.

[16] Glorot, X. et al. *Deep Sparse Rectifier Neural Networks*. Published in: *Aistats Volume 15*. 2011, 315-323.

[17] Thibodeau, Patrick. *Scientists, IT Community Await Exascale Computing*. Computerworld, 7 December 2009. <http://www.computerworld.com/article/2550451/computer-hardware/scientists--it-community-await-exascale-computers.html>

[18] Darling, Patrick. *Intel Equipped to Lead Industry to Era of Exascale Computing*. Intel Newsroom, 20 June 2011. <https://newsroom.intel.com/news-releases/intel-equipped-to-lead-industry-to-era-of-exascale-computing/>

7 Acknowledgement

This paper would not have been realizable without the effort done by Daniel Oehry by providing me with helpful tips and insights. He was kind enough to support and supervise me during the process of writing. Furthermore, I would also like to thank my brother, Noldi Wohlwend, for proofreading the final revision. Last but not least, the artificial intelligence, Rachel, shall not be forgotten for following the obvious and boring tasks she was given without asking any questions.

8 Declaration of Own Work

Hereby I declare that I have written this paper self-handedly using only the permitted tools. I especially assure that I have made all literal and analogous quotes from other corpora clearly visible as such.

Schellenberg, 15 April 2017

Konstantin Wohlwend